Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Peter Sanders
Moritz Kobitzsch, Dennis Schieferdecker

1. Übungsblatt zu Algorithmen II im WS 2011/2012

http://algo2.iti.kit.edu/AlgorithmenII.php
{kobitzsch,sanders,schieferdecker}@kit.edu

Musterlösungen

Aufgabe 1 (*Analyse: Kleinaufgaben*)

a) Geben Sie die wesentlichen Unterschiede –laut Vorlesung– zwischen einer (normalen) *Priority Queue* und einer adressierbaren *Priority Queue* an.

b) Vergleichen Sie die Laufzeit einer `merge` Operation für *Pairing Heaps* und *Binary Heaps*.

Musterlösung:

a) Im Gegensatz zur normalen *Priority Queue* erlaubt die adressierbare *Priority Queue* den direkten Zugriff auf beliebige Elemente über ein *Handle* h. Dieses wird von der `insert` Operation als Rückgabewert geliefert. Außerdem ermöglicht sie einige zusätzliche Operationen: `remove(h)`, `decreaseKey(h,k)`. Die `merge` Operation kann prinzipiell auch von normalen *Priority Queues* unterstützt werden, allerdings weniger effizient.

b) In einem *Pairing Heap* wird eine Menge von Bäumen gehalten. Eine `merge` Operation ist also in konstanter Zeit möglich, indem die jeweiligen Listen vereint werden und der `minPtr` auf das Minimum beider Wälder gesetzt wird. Im *Binary Heap* funktioniert dieses Vorgehen nicht. In der weit verbreiteten Arrayimplementierung gibt es mehrere mögliche Vorgehensweisen. Wenn die Anzahl der Elemente gegeben ist durch n_1 bzw. n_2, $n_1 < n_2$, ergeben sich folgende Laufzeiten:

   (a) Einfügen der kleineren Menge an Elementen: $O(n_1 \cdot \log(n_1 + n_2))$

   (b) Kompletter Neuaufbau: $O(n_1 + n_2)$

   Je nach Verteilung der Elemente können beide Möglichkeiten sinnvoll sein.

Aufgabe 2 (*Analyse: Laufzeitverhalten*)

a) Beweisen Sie allgemein für adressierbare *Priority Queues* die untere Laufzeitschranke von $\Omega(\log n)$ für `deleteMin` unter der Voraussetzung, dass `insert` konstante Laufzeit benötigt.

b) Warum muss diese untere Laufzeitschranke nicht gelten, wenn `insert` mehr Zeit benötigen darf?

Musterlösung:

a) Mit einer Laufzeit von $O(f(n))$ für `deleteMin` ließe sich in Zeit $O(nf(n)) + n \cdot O(1)$ vergleichsbasiert sortieren, indem man zuerst alle Elemente einfügt und dann eines nach dem anderen in aufsteigender Reihenfolge entnimmt. Für `deleteMin` in sublogarithmischer Zeit wäre das ein Widerspruch zur bekannten unteren Schranke für vergleichsbasiertes Sortieren von $\Omega(n \log n)$.

b) `insert` könnte nach jedem Aufruf eine aufsteigend sortierte Liste aller Elemente hinterlassen, mit deren Hilfe sich alle folgenden `min`- und `deleteMin`-Operationen in konstanter Zeit beantworten ließen.

Aufgabe 3 (*Analyse:* worst-case *Verhalten*)

a) Geben Sie einen Zustand eines *Fibonacci Heaps* an, für den die nächste `deleteMin` Operation $O(n)$ Laufzeit benötigt. Geben Sie auch die entsprechende `deleteMin` Operation an.

b) Beschreiben Sie, wie der von Ihnen angegebene Zustand erzeugt werden kann.

Musterlösung:

a) Wenn wir zu Beginn einer `deleteMin` Operation einen Wald betrachten, in dem jeder Knoten einen einzelnen Baum formt, so benötigt `deleteMin` lineare Zeit. Dies liegt an einer linearen Anzahl von `link` Operationen, die auf dem Wald durchgeführt werden müssen. Dabei wird in jedem Schritt die Anzahl der Wurzeln um eins verringert. Ist die Anzahl der Knoten eine Zweierpotenz, so wird dieser Vorgang solange ausgeführt bis genau ein Baum vorliegt. Die Anzahl der gesetzten Bits in der Binärdarstellung von n liefert die Menge der entstehenden Bäume. Da nur logarithmisch viele Bits gesetzt werden können hat die Operation deleteMin eine Laufzeit von $n - log(n)$ und ist somit linear.

   Anmerkung: Diese Argumentation funktioniert nur, weil wir hier volle Bäume betrachten aus denen kein Knoten per `cut` Operation entfernt wurde. Damit enhält ein Fibonacci Baum in Bucket k genau 2^k Knoten.

b) Ein derartiger Zustand kann durch eine Folge von n `insert` Operationen erreicht werden, da `insert` nur einen neuen Baum in den Wald einfügt.

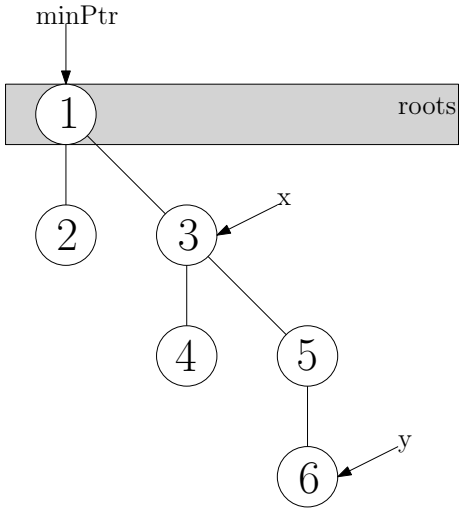
Aufgabe 4 (*Rechnen:* Pairing Heaps)

Gegeben sei ein *Pairing Heap* mit unten eingezeichnetem Zustand.

a) Geben Sie eine möglichst kurze Folge von Operationen an, die diesen Zustand erzeugt.

b) Führen Sie anschließend folgende Operationen auf dem Heap aus und zeichnen Sie den Zustand des Heaps nach jeder Operation:

- `deleteMin()`
- `insert(9)`
- `decreaseKey(x, 1)`
- `remove(y)`



Musterlösung:

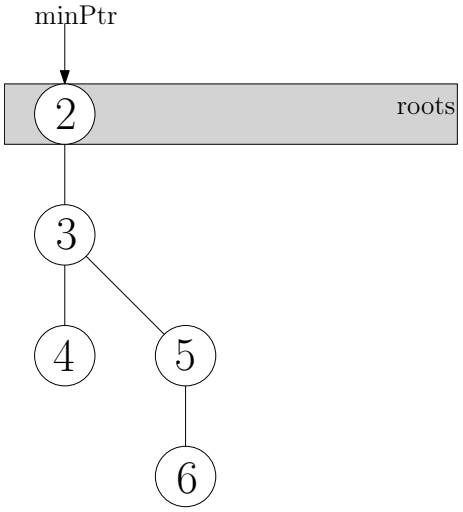
a) Die Folge

`insert(0)`, `x:=insert(3)`, `insert(4)`, `insert(5)`, `y:=insert(6)`, `deleteMin()`, `insert(0)`, `insert(1)`, `insert(2)`, `deleteMin(), insert(0), deleteMin()`

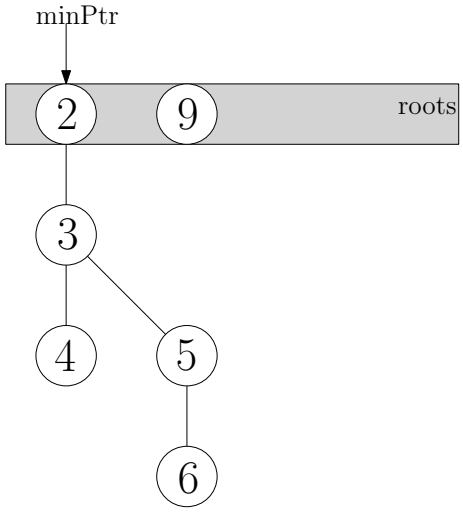
erzeugt den gegebenen Zustand.

Beachten Sie, dass die Definition des *Pairing Heaps* keine Aussage darüber macht, welche Elemente Nachbarn sind. Für diese Lösung und die der nächsten Teilaufgabe nehmen wir eine nach der Reihenfolge der Einfügungeoperationen sortierte Liste von Wurzeln und eine dadurch induzierte Nachbareigenschaft an.

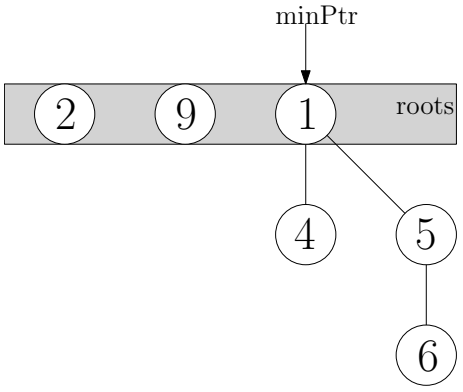
b) `deleteMin()`:



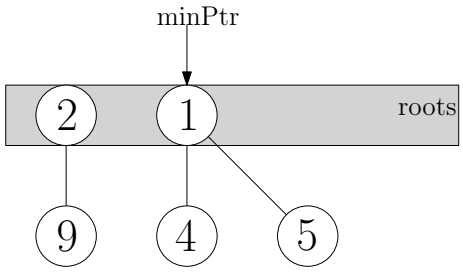
`insert(9)`:



`decreaseKey(x,1)`:



`remove(y)`:



Aufgabe 5 (*Entwurf: Datentypen*)

a) Erweitern Sie den Datentyp *Pairing Heap* um die Operation `increaseKey(h: Handle, k: Key)`. Ihre Operation sollte amortisiert $O(\log n)$ Laufzeit benötigen. Geben Sie Pseudocode an. Wie würden Sie bei einem *Binary Heap* vorgehen?

b) Entwerfen Sie einen Datentyp der die Operationen `insert` in $O(\log n)$, Median bestimmen in $O(1)$ und Median entfernen in $O(\log n)$ unterstützt. Eine Beschreibung in Worten ist ausreichend.

Musterlösung:

a) Lösche das Element aus der Datenstruktur ($O(\log n)$) und füge es mit dem geänderten Schlüssel wieder ein ($O(1)$):

```
1: function increaseKey(h : Handle, k : Key)
2:     remove(h)
3:     key(h) := k
4:     insert(h)
5: end function
```

Bei einem *Binary Heap* würde man zuerst den Schlüssel anpassen und anschließend eine `siftDown` Operation ausführen.

b) Aufbau des Datentyps:

- Speicherung des aktuellen Median in v.
- Verwendung einer Maximum *Priority Queue* (MaxPQ) für Elemente kleiner als v und einer Minimum *Priority Queue* (MinPQ) für Elemente größer als v.

Bestimmung des Median:
Frage v direkt ab: $O(1)$.

Löschen des Medians:
Ersetze den aktuellen Median v durch das oberste Element der größeren *Priority Queue* (bei Gleichheit verwende MinPQ): `deleteMin` in $O(\log n)$, z.B. mit *Fibonacci Heap*.

Einfügen eines Elements:
Füge das neue Element –in Abhängigkeit von v– in eine der *Priority Queues* ein: `insert` in $O(1)$. Füge v in die kleinere ein (bei Gleichstand verwende MaxPQ): `insert` in $O(1)$. Ersetze v durch das oberste Element der größeren *Priority Queue* (bei Gleichstand verwende MinPQ): `deleteMin` in $O(\log n)$, z.B. mit *Fibonacci Heap*.

Aufgabe 6 (*Entwurf: Anwendung*)

Für ein großes –und wir meinen ein wirklich großes– Fest sind Sie für die Bar zuständig. Für diese Aufgabe haben Sie Ihren eigenen Barroboter entworfen, der automatisch wunderbare Cocktails mischen kann. Die Zutaten dafür werden in großen Kanistern bereitgestellt. Doch genau hier ist das Problem: In all der Hektik des Abends müssen Sie darauf achten, dass kein Kanister leerläuft. Sie wollen den Abend allerdings auch so gut wie möglich genießen und nicht andauernd die Kanister überprüfen. Um dieses Problem zu umgehen, gibt es nur eine Lösung: Eine Nachfüllanzeige muss her! Leider gab es nur noch Anzeigen, die es erlauben eine einzelne Zeile darzustellen.
Es ist klar, dass auf der Anzeige die am dringensten benötigte Zutat angezeigt werden sollte. Ziel ist es also einen Algorithmus zu entwerfen, der die Anzeige immer aktuell hält.

a) Überlegen Sie sich, welche Datenstruktur Sie als Grundlage für Ihren Algorithmus verwenden wollen, um ihn effizient implementieren zu können. Sie können davon ausgehen, dass die für einen Cocktail benötigten unterschiedlichen Zutaten wesentlich weniger sind als die Gesamtmenge an vorhandenen unterschiedlichen Zutaten.

b) Entwerfen Sie eine Funktion `MixDrink( recipe )`, die auf Ihrer Datenstruktur operiert. Geben Sie Pseudocode an. Sie müssen dabei nur Ihre Datenstruktur aktualisieren. Sonstige Funktionen des Roboters müssen Sie nicht berücksichtigen.

c) Wenn ein Kanister gewechselt wird, muss ihre Datenbasis natürlich auch aktualisiert werden. Beschreiben Sie, welche Auswirkungen das Wechseln auf Ihre Datenstruktur hat.

Musterlösung:

a) Die Zutat, die auf dem Display angezeigt werden soll, ist die, die den geringsten Restvorrat vorweist. Zusätzlich sollen die Mengen aller Zutaten im laufenden Betrieb aktualisiert werden können. Dabei werden Zutaten in ihrer Menge immer reduziert. Die klassische Datenstruktur hierfür ist ein adressierbarer *Heap*.

b) Die Funktion `MixDrink`, wie gefordert, hat nur die Aufgabe die vorhandene Zutatenmenge aktuell zu halten.

```
1: function MixDrink(r : Recipe, q : Queue, d : Display)
2:     for ingredient i ∈ r do
3:         q.decreaseKey(i, amount(i))
4:     end for
5:     d.show(q.min(), amount(q.min()))
6: end function
```

c) Beim Wechsel eines Kanisters muss das Element aus dem *Heap* entfernt werden und mit vollem Kanisterwert wieder eingefügt werden. Dabei ist allerdings zu beachten, dass nicht zwingend das aktuell minimale Element gewechselt wird. Daher sollte eine Folge von Operationen durchgefürt werden. Als erstes sollte der Schlüssel der betroffenen Zutat auf $-\infty$ gesetzt werden. Danach kann ein `deleteMin` gefolgt von einem `insert` ausgeführt werden.